

UPDATE — A Stealthy Sniffer Detector

Part II

Jim Mellander

Introduction

Part 1 (in last month's *Information Security Bulletin*) of this two-part paper series discussed some of the issues surrounding the use of sniffer detectors. The previous part covered why they are needed, the current crop of sniffer detectors in widespread use, what motivated developing a new stealthy sniffer detector, and development problems and solutions. Part 2, the current and final part of this series, discusses customizing the response that UPDATE provides, adding encryption for a stealthier "footprint," supporting a wider range of platforms, installing and using this tool (including experiences that occurred from running it), and, finally, how to obtain a copy.

Customized Response

Recall from Part I that although UPDATE worked and delivered the basic functionality that was originally envisioned for it, it originally also had a few limitations. To address these issues required forging ahead on several fronts: porting continued to expand the range of Unix platforms supported, and several items on the "complaint list" were examined and addressed. Administrators wanted an easy way to customize the shell script without using a compiler; they were also concerned that the text of the shell scripts was visible in the binary version. These "complaint list" items are actually related, as we will see shortly.

The solution that I chose to correct these complaints was to set aside space in the binary with a recognizable signature, then actually patch the binary with the desired shell script. A large string was set aside as the command to be run by the system() function, filled with repeated instances of "PATCH_ME_HERE_UPDATE." A patch program was developed which looks for that signature in the binary, checks that an excessively large shell script will not overflow the buffer, and replaces the string data with the shell script.

I developed a script that standardized the most common reporting actions that I anticipated a system administrator would want. Any of the following options could be selected as desired:

- Report via email
- Report via pager
- Report via the syslog facility
- Turn off network interfaces
- Shutdown system

Each of these options generates a shell script fragment. Shell script fragments are in turn combined to create a script to implement these actions. The first three options, email address, pager PIN, and syslog logging level, prompt for further information. The last two options contain

a built-in delay, so that the previous options have a chance to function *before* network connectivity is dropped or the system is shutdown (which also, of course, also results in dropping network connectivity).

The shell script is displayed for the administrator, who can edit it to select customized actions. Because the default script has to work across multiple platforms, it includes a search path sufficient to cover all the supported platforms. The script uses only standard installed Unix commands. For instance, the shell command to turn off the network interfaces is implemented portably by entering:

```
sleep 5;netstat -rn | awk '$NF !~ /^lo*/ {print "ifconfig " $NF "
down&" }' | sh
```

When this command is executed, the interface names from the routing table are extracted and `ifconfig <NAME OF INTERFACE> down`¹ is executed on all interfaces except the loopback. Standard output and standard errors are redirected to `/dev/null` before the shell script is run, preventing error messages from being printed.² When the script is finalized, the binary is patched with the script. Once this operation is done once, the binary can be deployed on all similar systems in an organization with the same reporting action policy.

A Dilemma: How to Protect Security Measures

Running `strings` on the UPDATE binary displays the shell script being executed, a dead giveaway when the email message “Alert — Promiscuous Mode Detected” is embedded in the script. Although it seems unlikely that an attacker would scan the binary, the threat of someone doing this seemed troubling. For better or worse, UPDATE uses the “security by obscurity” paradigm by hiding security measures so that attackers are unaware of their presence.³ “Security by obscurity” is not always bad; in this case we have in effect at least added another barrier that an attacker must address to avoid detection. By obfuscating the command to be executed, we create more potential work for an attacker and also possibly defeat casual attacks.

Encryption of the shell script provides a solution. If the script is encrypted when it is embedded in the binary, then decrypted when executed, we can at least accomplish the goal of hiding the script from the `strings` command.

Many well-known, cryptographically strong algorithms with easy-to-use APIs could be used for this purpose. These, however, exceed the needs of UPDATE for several reasons:

¹ This is a command that shuts the interface (e.g., `eth0` in Linux) down.

² These messages would otherwise be printed due to the heading printed by the `netstat` command in some platforms.

³ In a sense, publicizing the UPDATE program in this paper defeats this paradigm, since attackers may now become aware of this software.

1. These packages are designed for secure communication of data, rather than simple obfuscation. As such, they provide more functionality than needed. They also expand the code size considerably.
2. Once the presence of UPDATE is detected by an attacker, the game is over. An attacker with root privileges can easily kill the process to stop the sniffing activity. The actual shell script executed when promiscuous mode is detected is in this sense irrelevant to an attacker, who can at least be confident that it notifies security personnel. The fact that a sniffer detector is running is thus in reality the critical information being protected. This information is fundamentally different from the type that cryptographic algorithms protect.

Simple Encryption with XOR

With these caveats in mind, I decided on a simpler, tinier, encryption scheme. UPDATE's cryptographic capabilities depend on the properties of the Boolean function XOR. If a value is XOR'd with any key (except 0), a different value is returned. If this returned value is again XOR'd with the same key, the original value is returned. In our application, the shell script is encrypted by XOR'ing each byte with a reproducible pseudorandom byte stream. To decrypt, the encrypted (using the term loosely) stream is again XOR'd with the same pseudorandom byte sequence. C provides the `rand()` function to generate random integers, as well as the `srand()` function to seed the `rand()` function with a known starting point. This seemed ideal for this encryption scheme.

Naturally, things become more complicated. Vendors have been known to change library functions such as `rand()` and `srand()` to improve randomness and to add functionality.⁴ I became concerned that a binary built on an older version of an operating system may fail (silently!) when an upgrade occurs. To address these concerns, simple implementations of `rand()` and `srand()` are included in UPDATE.

Another additional detail: If the shell script does not completely fit into the space allocated, the "signature" of the placeholder (`PATCH_ME_HERE_UPDATE`) could be revealed. Thus, the encryption phase encrypts all the way to the end of the placeholder. Decryption, however, stops when the first zero byte is decrypted signaling the end of the embedded shell script, since there is no need to continue.

Supporting More Platforms

AIX uses the BSD paradigm (with a "wrinkle"), while Linux is an evolving operating system, with differences primarily based in the kernel. We will start with AIX.

⁴ Frankly, the feared behavior by vendors has not generally manifested itself in practice. During development of critical security software, nevertheless, heeding paranoid inclinations should be considered desirable.

AIX

As with BSD-Unix, an `ioctl()` call (`SIOCGIFFLAGS`) retrieves the flags for a known interface, and the `IFF_PROMISC` bit reliably indicates the state of the interface. Readers will remember that the list of installed interfaces is derived by executing the `SIOCIFCONF` `ioctl()`, which fills a buffer with a list of the interfaces installed on the system. Unfortunately, on AIX, this buffer came back filled with garbage, rather than useful information. I was unable to determine the cause of this phenomenon, although, of course, it is quite possible that it was a “short-circuit between the ears.” AIX being a minor platform in our installation, I decided to “punt”: the AIX code simply has a hard-coded list of standard interfaces to step through. Although a kludged scheme, in practice it has proven successful. For systems with additional interfaces, the code will need to be modified to accommodate the additional interfaces. For reference, the interfaces checked by UPDATE on AIX are: `en0`, `en1`, `ent0`, `ent1`, `et0`, `et1`, `fi0`, and `fi1`. Aside from this wrinkle, the standard BSD code is used. UPDATE has been successfully compiled and used on AIX 3.5.x and AIX 4.2.x.

Linux

The popularity of Linux, a freely available Unix implementation with a large software base, motivated me to port UPDATE to this operating system. UPDATE has been successfully compiled and used on the Linux 2.0 kernel, the Linux 2.2 kernel, and Sparc Linux (2.2 kernel). The next section of this paper discusses the Linux implementation of UPDATE.

Special Considerations with Linux

Although it uses a BSD-style socket interface, Linux has an independently developed networking implementation. Up to the 2.0 series kernels, the BSD-style code for UPDATE worked perfectly. When the 2.2 kernel came out, I began receiving complaints that UPDATE was not working. After verifying that this was the case, I began solving this problem. One advantage of Linux is the readily available source code for all aspects of the system, allowing inspection of the internals of the networking code.

I found that the Linux 2.2 kernel has two sets of flags, the standard flags for the interface, and a set of “global” flags that contain the “global” promiscuous and multicast bits. For some reason that my research failed to uncover, when the `SIOCGIFFLAGS` `ioctl()` is made, the actual promiscuous and multicast bits of the interface (which are correct!) are overlaid with the “global” bits. The purpose of those “global” flags remains a mystery.

My initial solution (which proved unsatisfactory) was to simply modify the kernel code to actually return the standard flags in response to the `SIOCGIFFLAGS` `ioctl()`. This worked fine with no apparent difficulties. However, I was uncomfortable with the notion of requiring users to patch and rebuild the kernel to use UPDATE. I wanted to be able to use a standard kernel distribution.

Fortunately, the source code provides the solution. The kernel variable `dev_base` points to a linked list of the network interfaces installed on the system. Stepping through this linked list and reading the structure data for each interface allows the standard flags to be accessed. This linked list always starts with the loopback interface, and is terminated with a null pointer.

When first programming the code, I did not understand that the kernel symbols for the currently running kernel are easily accessible via the `/proc/ksyms` file. Instead, UPDATE looks for the standard locations of the `System.map` file, then reads and parses the file for the `dev_base` symbol. To protect against the `System.map` file being old or obsolete, the pointer is checked for sanity before use. Once I became aware of `/proc/ksyms`, I realized that the sanity checking is unnecessary, but I left it in place anyway because it was already there and did not hurt anything. It also provided a double check that the code was correct. Once `dev_base` is determined, the first item on the list is skipped, since it is always the dummy loopback interface, then the other interfaces are checked until promiscuous mode is found or a null pointer is encountered.

Vmware: A Wrinkle

Vmware is a popular Linux application. Through virtualization techniques, vmware allows Windows operating systems to run unmodified as “guest” operating systems under Linux. To run unmodified Windows networking applications while still supporting Linux, vmware creates a “virtual” networking interface with a different address from that used by Linux. In order to implement this, vmware puts the interface into promiscuous mode. This will cause UPDATE to fire off a false alarm. Unfortunately, turning off promiscuous mode causes vmware networking to fail. Thus, the only solution seems to be to ignore the promiscuous mode flag while vmware is running. Unfortunately, this creates an opportunity for an attacker to “slide under the UPDATE radar” if an attacker is aware of this fact.

Rather than coding special cases such as this in the base code, I decided to incorporate the vmware detection code in the shell script that is executed. The `ps` command is executed, and grepped for vmware. If the string is detected, the shell script immediately exits without taking any action. Although a slight inefficiency is introduced by performing this recurring action which merely exits, it seemed to be the easiest method for dealing with this issue. It also provided a model for other applications that may exhibit the same behavior.

Other Platforms

AIX and Linux are not the only platforms to which UPDATE has been ported. Other platforms to which UPDATE has been ported include IRIX 6.5.x, Solaris 2.5 and above, Sun OS 4, Digital Unix (now Tru64 Unix), HP/UX 10, FreeBSD and others. All of the platforms not specifically described in this two-part series use the BSD style networking code, and required no additional modifications.

Disabling Promiscuous Mode

One frequently requested option was the ability to turn off promiscuous mode on interfaces on which it is enabled. On some systems (those for which promiscuous mode detection is not difficult!), it was easy to turn it off using the documented `SIOCSIFFLAGS` call to set the promiscuous bit off. On the more difficult systems, turning off promiscuous mode is as difficult as discovering it in the first place! In Solaris, for instance, the per-stream information must be rewritten back to the kernel to turn off promiscuous mode for that stream. I was unable to get this feature to work in other platforms (Solaris X86, Linux 2.2); in some cases, networking instability resulted. This feature is thus disabled in these platforms.

Since the option to disable promiscuous mode is performed in the `UPDATE` binary, it is not coded as a response via a shell-script. It is instead hard-coded into `UPDATE` for those platforms on which it works. When it works, the results are quite dramatic — running `snoop` as root on Solaris will cause all network traffic for the box to start scrolling on the display terminal. When `UPDATE` turns off promiscuous mode, `snoop` keeps working, but only local (and broadcast traffic) are displayed. Unless an attacker is familiar with normal network traffic generated by a system, it is quite possible that an attacker may conclude that there is nothing of interest on the network, anyway. By then, the alert system administrator will be aware of the sniffing activity.

Installing and Testing UPDATE

The next section of this paper covers how to install and test `UPDATE`. First, I'll describe how to install this tool.

Installation

Installing `UPDATE` requires several steps. First, download the distribution from <ftp://lassie.lbl.gov/UPDATE/>, then `gunzip` and `untar` the distribution. Pending licensing issues, only a binary distribution is provided. To install the precompiled binaries, go to the `dist` sub-directory, and copy the `UPDATE.in.*` file for your platform to `UPDATE.in`. Then copy the `patcher.*` file for your distribution to `patcher`. Patch the binary by first entering:

```
./patch.sh
```

The five previously discussed reporting methods (e.g., email, pager, and so forth) will now be displayed. You can select any combination of these five. Enter the number of the option you desire, then press `<Enter>`. You can enter multiple options separated by any characters (or none at all) on one line. For instance, entering “12345” causes all five options to be selected, as will “1,2,3,4,5.” The menu will be redisplayed with an asterisk (*) next to each selected option. To deselect an option, simply enter the option number again. When finished, press 0 to continue with the configuration. Depending on the options selected, you must now enter additional information:

- For the email configuration, you will be prompted for the email address.
- For pager configuration, enter the PIN of the pager to “beep.” The paging format will be specific to each organization and thus requires modification. The pager configuration uses a specific email address that, when emailed with a PIN and message, contacts the pager company and arranges to send the message to the alpha pager corresponding to the PIN. Some administrators may enjoy early morning pages, but I found it as just as useful to send an email, then shut off the network interface, so that the system is “locked down” until the administrator gets to the system console.
- For `syslog` configuration, enter the logging priority or press `<Enter>` for the default of `auth.crit.` (Enter `man logger` for more info).
- There are no options for the Shutdown Network Interface Option. It simply waits five seconds to allow email, etc. to be delivered, then shuts down all network interfaces except loopback. Keeping loopback up allows local networking services (such as X-Windows if you are not using Unix domain sockets) to function normally, but blocks all remote access.
- Likewise, there are no options for the System Shutdown Option. It waits five seconds, then executes `init 0`, which will shut down the system.

Note that the options are executed in numerical order, so that you can send email, then page the administrator, then post to `syslog`, then shut down the network, then shut down the system itself.

A shell command that is displayed prior to the actual patching process is constructed from the selected options. At this point you are given the option of editing the script. The editor to be run is defined by the `EDITOR` environment variable; the default is `vi`. When the command is as you want it, type “0” to patch the binary. The shell command is run by `sh`, which thus requires that you use the appropriate semantics.

The `patcher` program embeds and encrypts the shell script directly in the binary. When complete, your custom `UPDATE` binary will be in the current directory, and can be installed in the startup binary directories by copying it to `/sbin` or `/usr/sbin`, depending on where they are normally installed. If you already have an `UPDATE` program, rename it to “`UPDATE0`.”

Next, I ensure that this program is well-hidden. The permissions and ownerships should be the same as other programs in the install directory. Set the timestamp back by entering:

```
chown bin update
chgrp bin update
touch -r wall update
```

I use the `wall` program as a reference timestamp, but you should look in the install directory for a program with an innocuous time stamp.

Testing

To initially test UPDATE, simply run it as root-equivalent, using the same shell that is used during the boot process (usually `/bin/sh`, `/bin/ksh`, or `/bin/bash`). UPDATE will immediately put itself in the background and start doing its magic. To make it run on system startup, add to your `/etc/rc*` files. UPDATE ignores all signals, so to turn it off you must execute a `kill -9`.

It is also wise to rename UPDATE. If attackers identify that a program named “UPDATE” is running, they are likely to do something undesirable (especially if they have root access). Fortunately, there is no reason the program needs to be named “UPDATE”; any innocuous name will do. UPDATE can even be started via `cron` (using the `-once` flag) or manually.

After completing initial testing and making any changes dictated by the test results, I suggest performing a final test of UPDATE before you rely on it. Run `tcpdump`, `snoop`, `netsnoop`, or another command to sniff the network. Within three minutes you should receive email (provided, of course, that you have configured the email option) notifying you that the system has a security violation. If the sniffer is running when UPDATE is started, you will get an immediate response, so testing will in this case be faster.

Command Line Options

Various command line options to UPDATE are available. The `-once` option, for example, causes UPDATE to run one time, then exit, something that is valuable for testing purposes. UPDATE also allows up to two numeric arguments, namely the time delay in seconds between `sync` calls and sniffer detection, respectively. Defaults (30 seconds for `sync`, 180 seconds for sniffer detection) are used whenever arguments are omitted or invalid. For users with a large RAID system or other method for flushing the buffers, `sync` calls can be effectively disabled by setting `sync` to a suitably large number. A setting of 32000000 will, for example, create a time delay between `syncs` of more than a year. You can also enter “UPDATE 86400 300” to run `sync` every day (86400 seconds) and sniffer detection every 300 seconds.

Conclusion

UPDATE has become the standard sniffer detector software in the organization for which I previously worked. It is an important part of an integrated security program that includes firewall deployment, intrusion detection, vulnerability scanning and security awareness training. UPDATE has successfully detected and deflected many attacks in which sniffers were installed without authorization. Support from the user community has been uniformly positive and helpful — many of the enhancements and improvements have in fact been driven by dedicated system administrators. Any suggestions or questions from readers are also welcome. I hope you find UPDATE as useful as many others have.

Acknowledgements

Many people have contributed to making UPDATE successful, but the following individuals⁵ are worthy of special mention: David Curry (IBM) (for writing the original ifstatus program), Neal Mackanic for suggestions on improvement, Dave Temple for the idea of shutting down the network interface, Ken Underhill for user interface suggestions, Dave Martini for allowing me access to some of his machines to support more interfaces under Solaris, Dave Williams for access to a Solaris x86 system, Jimmy Guse for information on FreeBSD, and all beta testers.

⁵ I apologize to anyone whose name I have omitted. If you email me, I'll include you in all subsequent publications and reports about UPDATE.